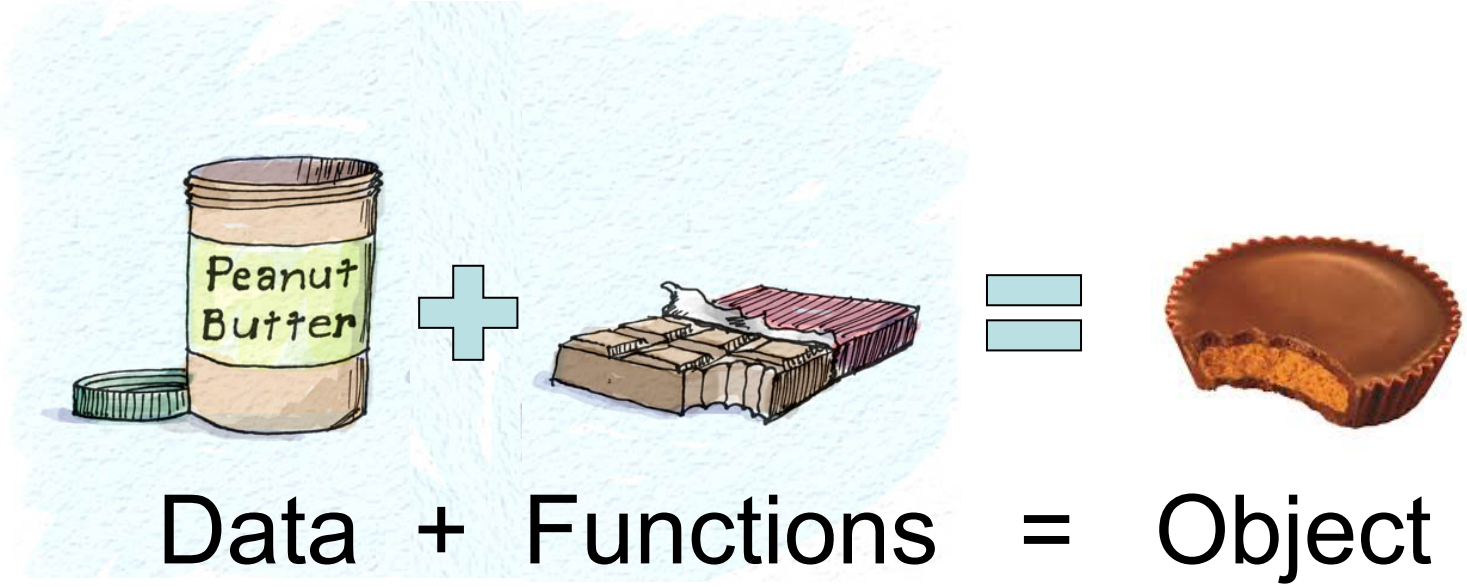
Object-Oriented Programming (OOP)

It's like peanut butter and chocolate.
Each is great, but they're better together.



Data + Functions = Object

Classes and Objects

An object-oriented programming language allows you to build your **own customized types** of variables.



(1) A *class* is a **type**

(2) An *object* is one such **variable**.

(instance)



There will typically be MANY objects of a single class.



Classes and Objects

Customizing Python



(1) A *class* is a **type**

(2) An *object* is one such **variable**.

(instance)



There will typically be MANY objects of a single class.



Everything in Python is an **object**!

Its capabilities depend on its **class**.



functions

"methods"



type

What's more, you can build your own...

Everything is an object!

Take strings, for example:

```
>>> s = str(42)
>>> type(s)
<type 'str'>
>>> dir(s)
```

This calls the **`str` *constructor***.

Shows the type of **`s`** is **`str`**

Shows all of the methods (functions) of **`s`**

```
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'_formatter_field_name_split', '_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Let's try some!

Objects

Like a list, an object is a container, but much more customizable:

(1) Its data elements have *names chosen by the programmer*

(2) An object contains its own functions, called ***methods***

(3) In its methods, objects refer to themselves as **`self`**

(4) Python signals special methods with double underscores:

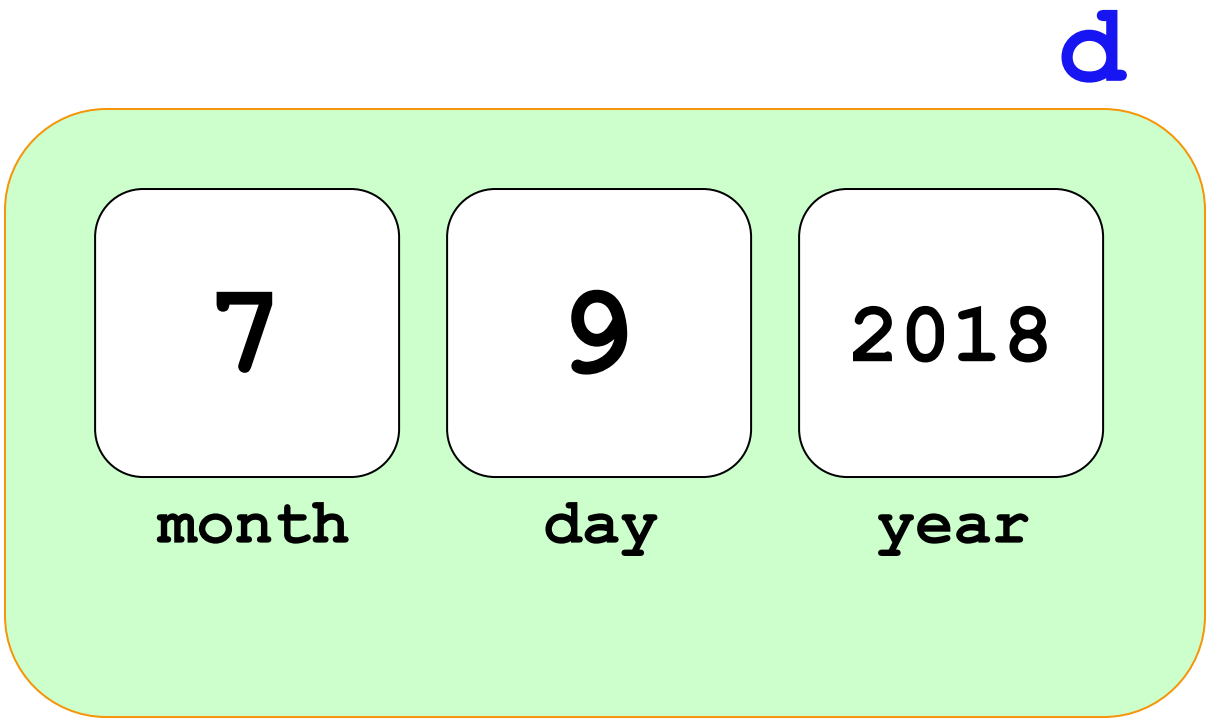
`__init__` is called the ***constructor***; it creates new objects

`__repr__` tells Python how to print its objects



I guess we should doubly ***underscore*** *these two methods!*

A `Date` object, `d`



memory location == 47245108030176

The `Date` class

```
class Date(object):
    """A user-defined data structure that
       stores and manipulates dates."""
```

This is the start of a new type called Date
It begins with the keyword **class**

This is the **constructor** for Date objects
As is typical, it assigns its arguments to the data members.

```
def __init__(self, mo, dy, yr):
    """Construct a Date with
       the given month, day, and year."""
    self.month = mo
    self.day = dy
    self.year = yr
```

These are data members–
they are the information
inside every Date object.

Date

This is a **class**. It is a user-defined data type that you'll build this afternoon

```
>>> d = Date(7,9,2018)
```

Constructor!

Add a print statement to see that this is OUR OWN code…

```
>>> d.month
4
>>> d.day
4
```

d contains data members named **day**, **month**, and **year**

```
>>> d
04/04/2017
```

The repr!

the **repr**esentation of an object of type Date

```
>>> d.isLeapYear()
False
```

The **isLeapYear** method returns **True** or **False**. How does it know ***what year to check***?

The `Date` class

```
class Date(object):
    """A user-defined data structure that
       stores and manipulates dates."""

    def __init__(self, mo, dy, yr):
        """Construct a Date with
           the given month, day, and year."""
        self.month = mo
        self.day = dy
        self.year = yr

    def __repr__(self):
        """Display a date in a nice format."""
        s = "{:02}/{:02}/{:04}".format(self.month, self.day, self.year)
        return s
```

This is the **repr** for Date objects
It tells Python how to print these objects.

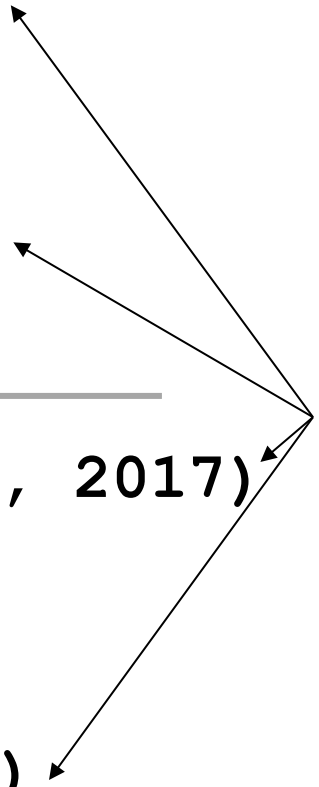
Why **`self`** instead of **`d`** ?

`self` is the variable calling a method

```
>>> d = Date(4, 4, 2017)
>>> print d
04/04/2017

>>> d.isLeapYear()
False
```

```
>>> ny = Date(1, 1, 2017)
>>> print ny
01/01/2017
>>> ny.isLeapYear()
False
```



These methods need access to the object that calls them: it's **`self`**

The `Date` class

```
class Date(object):
    """A user-defined data structure that
       stores and manipulates dates."""

    def __init__(self, mo, dy, yr):
        """Construct a Date with
           the given month, day, and year."""
        self.month = mo
        self.day = dy
        self.year = yr

    def __repr__(self):
        """Display a date in a nice format."""
        s = "{:02}/{:02}/{:04}".format(self.month, self.day, self.year)
        return s

    def isLeapYear( self ):
        """Anyone know the rule?"""
```

which are leap years?

2.2.1 What years are leap years?

The Gregorian calendar has 97 leap years every 400 years:

> Every year divisible by 4 is a leap year.
> However, every year divisible by 100 is not a leap year.
> However, every year divisible by 400 is a leap year after all.

So, 1700, 1800, 1900, 2100, and 2200 are not leap years. But 1600, 2000, and 2400 are leap years.

```
class Date(object):
    def __init__(self, mo, dy, yr): # (constructor)
    def __repr__(self): # (for printing)
```

```
    def isLeapYear(self):
        """Here it is in all its glory"""
        if self.year % 400 == 0: return True
        elif self.year % 100 == 0: return False
        elif self.year % 4 == 0: return True
        return False
```

== vs. equals



UFO license
Area 51, NV

What **id** is on your Date?

```
>>> wd = Date(11, 12, 2013)
>>> wd
11/12/2013
```

this constructs a different Date

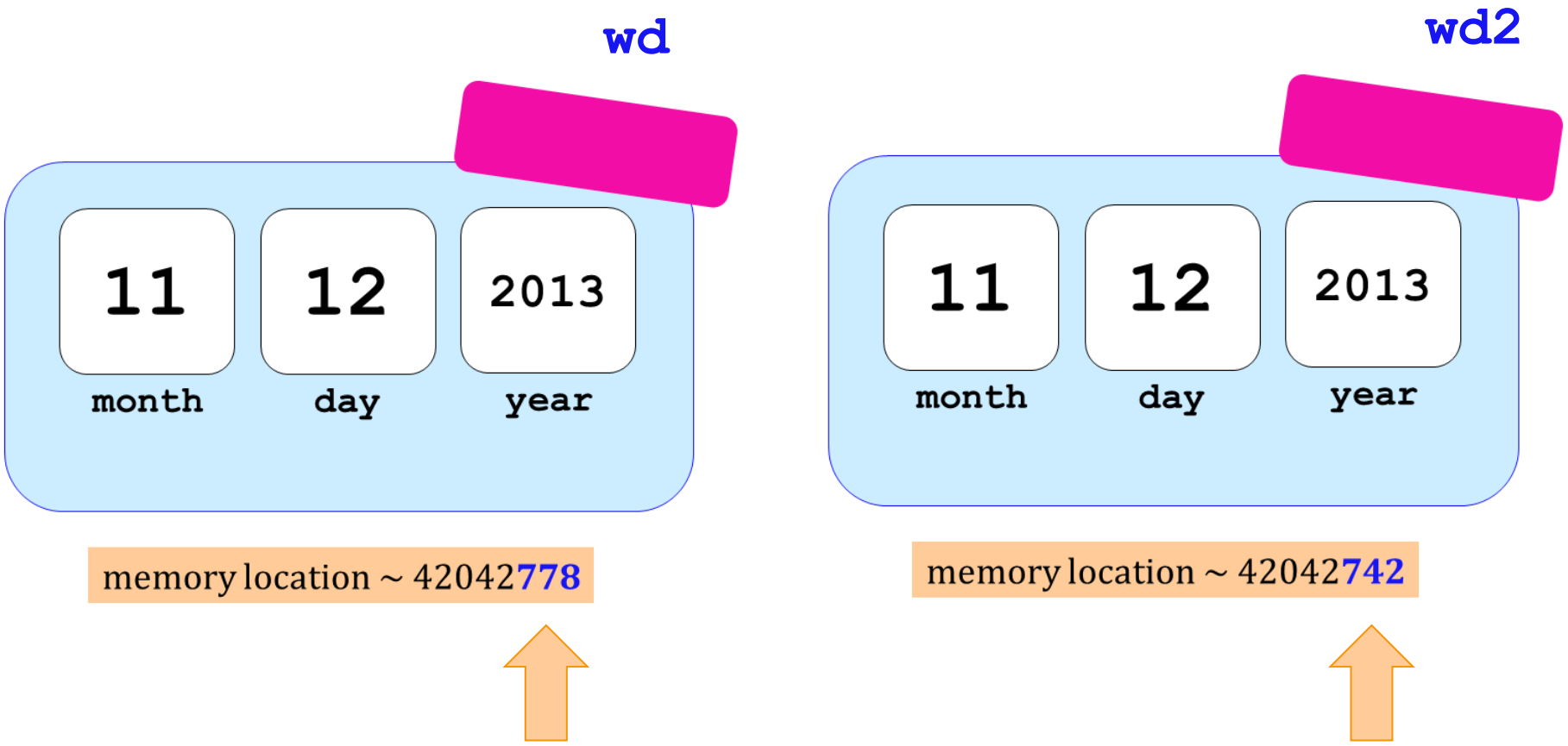
```
>>> wd2 = Date(11, 12, 2013)
>>> wd2
11/12/2013
```

```
>>> wd == wd2
False
```

How can this be False ?

Python objects are handled by reference…
== compares references!

Two `Date` objects:



== compares **memory locations**, not contents

originals underneath...

== vs. equals



UFO license
Area 51, CA

What date is on your **id**?
What **id** is on your Date?

```
>>> wd1 = Date(11, 12, 2013)
>>> wd1
11/12/2013
```

this constructs a different Date

```
>>> wd2 = Date(11, 12, 2013)
>>> wd2
11/12/2013
```

```
>>> wd1.equals(wd2)
True
```

Python objects are handled by reference…
.equals compares contents

equals

```
class Date(object):

    def __init__(self, mo, dy, yr): …
    def __repr__(self): …
    def isLeapYear(self): …

    def equals(self, d2):
        """Returns True if self and d2
           represent the same date;
           False otherwise."""

        if self.year == d2.year and \
           self.month == d2.month and \
           self.day == d2.day:
                return True
        else:
                return False
```

How many Dates are here?



Which one is **self**ish?

To use this, write `d.equals(d2)`

equals

```
class Date(object):

    def __init__(self, mo, dy, yr): …
    def __repr__(self): …
    def isLeapYear(self): …

    def __eq__(self, d2):
        """Returns True if self and d2
           represent the same date;
           False otherwise."""

        if self.year == d2.year and \
           self.month == d2.month and \
           self.day == d2.day:
                return True
        else:
                return False
```

Redefined for *our convenience*!

To use this, write `d == d2`

DIY operators ...

__eq__(self, other) defines the equality operator, ==
__ne__(self, other) defines the inequality operator, !=
__lt__(self, other) defines the less-than operator, <
__gt__(self, other) defines the greater-than operator, >
__le__(self, other) defines the less-or-equal-to operator, <=
__ge__(self, other) defines the gr.-or-equal-to operator, >=

__add__(self, other) defines the addition operator, +
__sub__(self, other) defines the subtraction operator, -

... and many more! Use dir('')

there are two underscores
on each side here

I should **underscore** this unusual syntax!



More operators!

arithmetic

```
__add__(self, other)¶   +
__sub__(self, other)    -
__mul__(self, other)    *
__matmul__(self, other) @
__truediv__(self, other)
__floordiv__(self, other)
__mod__(self, other)
__divmod__(self, other)
__pow__(self, other[, modulo])
__lshift__(self, other)
__rshift__(self, other)
__and__(self, other)
__xor__(self, other)
__or__(self, other)
```

Booleans

```
__lt__(self, other)
__le__(self, other)
__eq__(self, other)
__ne__(self, other)
__gt__(self, other)
__ge__(self, other)
```

in-place arithmetic

```
__iadd__(self, other)   +=
__isub__(self, other)   -=
__imul__(self, other)¶  *=
__imatmul__(self, other) @=
__itruediv__(self, other)
__ifloordiv__(self, other)
__imod__(self, other)
__ipow__(self, other[, modulo])
__ilshift__(self, other)
__irshift__(self, other)
__iand__(self, other)
__ixor__(self, other)
__ior__(self, other)
```

https://docs.python.org/3/reference/datamodel.html#special-method-names

Exercise today

Add these to your **`Date`** class!

```
yesterday(self)
tomorrow(self)
addNDays(self, N)
subNDays(self, N)
isBefore(self, d2)
isAfter(self, d2)
diff(self, d2)
dow(self)
```

```
class Date(object):

    def isBefore(self, d2):
        """True if self is before d2, else False."""
        if self.year    < d2.year:
            return True
        elif self.month < d2.month:
            return True
        elif self.day   < d2.day:
            return True
        return False
```

Challenge #1

If

`d` prints as 4/1/2018
`d2` prints as 4/4/2018

What does `d.isBefore(d2)` return?
Which of the *4* return statements is used?
Is this the *correct* value?

Challenge #2

Find *different* dates, d and d2, for which `d.isBefore(d2)` returns an INCORRECT value...

This Date is Late!



Extra! Above, show how to *fix* the **isBefore** method ...

isBefore

```
class Date(object):

    def isBefore(self, d2):
        """True if self is before d2, else False"""
        if self.year    < d2.year:
            return True
        elif self.month < d2.month and self.year == d2.year:
            return True
        elif self.day   < d2.day   and self.year == d2.year \
                                   and self.month == d2.month:
            return True
        return False
```

__lt__ <

*I want **LESS** !*



```
class Date(object):

    def __lt__(self, d2):
        """If self is before d2, this should
           return True; else False."""

        if self.isBefore(d2):
            return True
        else:
            return False
```

__lt__ <




```
class Date(object):

    def __lt__(self, d2):
        """This is less than most code!"""
        return self.isBefore(d2)
```

Date's *purpose*...?!




```
>>> d = Date(4, 4, 2018)
>>> print d
4/4/2018
```

always create with the **CONSTRUCTOR** ...

```
>>> d.tomorrow()
>>> print d
4/5/2018
```

the **tomorrow** method returns nothing at all. Is it doing anything?

d has changed!

```
d += 1
```

```
class Date:
```




```
    def tomorrow(self):
        """Moves the self date ahead 1 day."""

        DIM = [0,31,28,31,30,31,30,31,31,30,31,30,31]

        self.day += 1

        if
```

first, add 1 to `self.day`

DIM looks pretty bright to me!



test if we have gone "out of bounds!"

then, adjust the month and year, only if needed

Don't return anything. This **CHANGES** the date object that calls it.

```
class Date(object):

  def tomorrow(self):
    """Moves the self date ahead 1 day."""

    DIM = [0,31,28,31,30,31,30,31,31,30,31,30,31]

    self.day += 1       # add 1 to the day!

    if self.day > DIM[self.month]:    # check day
        self.month += 1
        self.day = 1

        if self.month > 12:           # check month
            self.year += 1
            self.month = 1
```



Extra how could we make this work for leap years, too?

```
class Date(object):

  def tomorrow(self):
    """Moves the self date ahead 1 day."""
```



better as a *variable*!

```
    DIM = [0,31,fdays,31,30,31,30,31,31,30,31,30,31]

    self.day += 1       # add 1 to the day!

    if self.day > DIM[self.month]:    # check day
        self.month += 1
        self.day = 1

        if self.month > 12:           # check month
            self.year += 1
            self.month = 1
```

```
class Date(object):

  def tomorrow(self):
    """Moves the self date ahead 1 day"""
    if self.isLeapYear(): fdays = 29
    else: fdays = 28

    DIM = [0,31,fdays,31,30,31,30,31,31,30,31,30,31]

    self.day += 1       # add 1 to the day!

    if self.day > DIM[self.month]:    # check day
        self.month += 1
        self.day = 1

        if self.month > 12:           # check month
            self.year += 1
            self.month = 1
```

